
Proceedings Paper:
Nightingale, Peter orcid.org/0000-0002-5052-8634, Spracklen, Patrick and Miguel, Ian James (2015) Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row. In: Pesant, Gilles, (ed.) Principles and Practice of Constraint Programming:21st International Conference, CP 2015, Cork, Ireland, August 31 -- September 4, 2015, Proceedings. Lecture Notes in Computer Science . Springer , Netherlands , pp. 330-340.

https://doi.org/10.1007/978-3-319-23219-5_23

Automatically Improving SAT Encoding of Constraint Problems through Common Subexpression Elimination in Savile Row

Peter Nightingale, Patrick Spracklen, and Ian Miguel

School of Computer Science, University of St Andrews, St Andrews, UK
{pwn1, jlps, ijm}@st-andrews.ac.uk

Abstract. The formulation of a Propositional Satisfiability (SAT) problem instance is vital to efficient solving. This has motivated research on preprocessing, and inprocessing techniques where reformulation of a SAT instance is interleaved with solving. Preprocessing and inprocessing are highly effective in extending the reach of SAT solvers, however they necessarily operate on the lowest level representation of the problem, the raw SAT clauses, where higher-level patterns are difficult and/or costly to identify. Our approach is different: rather than reformulate the SAT representation directly, we apply automated reformulations to a higher level representation (a constraint model) of the original problem. Common Subexpression Elimination (CSE) is a family of techniques to improve automatically the formulation of constraint satisfaction problems, which are often highly beneficial when using a conventional constraint solver. In this work we demonstrate that CSE has similar benefits when the reformulated constraint model is encoded to SAT and solved using a state-of-the-art SAT solver. In some cases we observe speed improvements of over 100 times.

1 Introduction

The Propositional Satisfiability Problem (SAT) is to find an assignment to a set of Boolean variables so as to satisfy a given Boolean formula, typically expressed in conjunctive normal form [4]. SAT has many important applications, such as hardware design and verification, planning, and combinatorial design [14]. Powerful, robust solvers have been developed for SAT employing techniques such as conflict-driven learning, watched literals, restarts and dynamic heuristics for backtracking solvers [15], and sophisticated incomplete techniques such as stochastic local search [22].

The formulation of a SAT problem instance is vital to efficient solving. This has motivated research on preprocessing [27, 7], and inprocessing [12] where reformulation of the SAT instance is interleaved with solving. Both techniques are highly effective in extending the reach of SAT solvers, however they necessarily operate on the lowest level representation of the problem, the raw SAT clauses, where higher-level patterns are difficult and/or costly to identify.

Our approach is different: rather than reformulate the SAT representation directly, we apply automated reformulations to a higher level representation of the original problem. An increasingly popular means of deriving SAT formulations is by taking a constraint model and employing a set of automated encoding steps to produce an equivalent

SAT formulation [28]. Constraint satisfaction is a formalism closely related to SAT in which we seek an assignment of values to decision variables so as to satisfy a set of constraints [21]. Constraint modelling languages typically support decision variables with richer domains and a richer set of constraints than the CNF used with SAT. Hence, an input problem can be expressed conveniently in a higher level constraint language, while employing efficient SAT solvers to find solutions.

Common Subexpression Elimination (CSE) is a very well established technique in compiler construction [5]. In that context the value of a previously-computed expression is used to avoid computing the same expression again. Shlyakhter et al [23] exploited identical subformulae during grounding out of quantified Boolean formulae. Similarly, it is a useful technique in the automatic improvement of constraint models, where it acts to reduce the size of a constraint model by removing redundant variables and constraints [10, 20, 19, 1]. This in turn can create a stronger connection between different parts of the model, resulting in stronger inference and reduced search during constraint solving.

Earlier work applied CSE directly to SAT formulations, with limited success [29]. Herein we establish the success of an alternative approach in which CSE is applied to a constraint model prior to SAT encoding. We apply CSE to a constraint problem instance expressed in the constraint modelling language ESSENCE′, which includes integer (as well as Boolean) decision variables, a set of infix operators on integer and Boolean expressions, and various global constraints and functions [20]. The reformulated constraint model is automatically encoded to SAT using the SAVILE ROW system, yielding substantial improvements in SAT solver runtime over encoding without CSE.

Our method has the advantage of allowing us to exploit patterns present in the high level description of the problem that are obscured in the SAT formulation, and so very difficult to detect using SAT pre/inprocessing approaches. As a simple example, a decision variable in a constraint model typically requires a collection of SAT variables and clauses to encode. If, via CSE, we are able to reduce two such variables to one then the set of Boolean variables and clauses to encode the second variable will never be added to the SAT formulation. Performing the equivalent step directly on the SAT formulation would require the potentially very costly step of identifying the structure representing the second variable then proving its equivalence to the structure encoding the first.

In performing CSE on a constraint model preparatory to SAT encoding, we have modified and enhanced existing constraint model CSE approaches to take into account that the eventual target is a SAT formulation. Firstly the set of candidate expressions for CSE differs when the target is SAT. Secondly, implied constraints that are added to elicit common subexpressions are removed following CSE if they are unchanged. In addition we describe for the first time an identical CSE algorithm that is independent of general flattening, allowing flexibility to extract common subexpressions would not ordinarily be flattened and to control the order of CSE.

2 CSE for SAT Encoding

The simplest form of CSE that we consider is Identical CSE, which extracts sets of identical expressions. Suppose $x \times y$ occurs three times in a model. Identical CSE would introduce a new decision variable a and new constraint $x \times y = a$. The three original

Algorithm 1 Identical-CSE(AST, ST)

Require: AST: Abstract syntax tree representing the model
Require: ST: Symbol table containing CSP decision variables
1: *newcons* ← empty list {Collect new constraints}
2: *map* ← empty hash table mapping expressions to lists
3: populateMap(AST, *map*)
4: **for all** *key* in *map* in decreasing size order **do**
5: *ls* ← *map*(*key*) {*ls* is a list of identical AST nodes}
6: *ls* ← filter(isAttached, *ls*) {Remove AST nodes no longer contained in AST or *newcons*}
7: **if** length(*ls*) $>$ 1 **then**
8: *e* ← head(*ls*)
9: *bnds* ← bounds(*e*)
10: *aux* ← ST.newAuxVar(*bnds*)
11: *newc* ← (*e* = *aux*) {New constraint defining *aux*}
12: *newcons*.append(*newc*)
13: **for all** $a \in ls$ **do**
14: Replace a with copy(*aux*) within AST or *newcons*
15: AST ← AST $\wedge$ fold($\wedge$, *newcons*)

Algorithm 2 populateMap(*A*, *map*)

Require: *A*: Reference to an abstract syntax tree
Require: *map*: Hash table mapping expressions to lists
1: **if** *A* is a candidate for CSE **then**
2: Add *A* to list *map*[*A*]
3: **for all** *child* $\in$ *A*.Children() **do**
4: populateMap(*child*, *map*)

occurrences of $x \times y$ would be replaced by a. In SAVILE ROW, Identical CSE is implemented with Algorithm 1. Andrea Rendl's Tailor [10, 20] and MiniZinc [26, 13] also implement Identical CSE, however (in contrast to Tailor and MiniZinc) our algorithm is not tied to the process of flattening nested expressions into primitive expressions supported directly by the constraint solver. This is advantageous because it allows us to identify and exploit common subexpressions in expressions that do not need to be flattened. The SMT solver CVC4 merges identical subtrees in its abstract syntax tree [3]. It is not clear whether this affects the search or is simply a memory saving feature.

The first step is to recursively traverse the model (by calling Algorithm 2) to collect sets of identical expressions. Algorithm 2 collects only expressions that are candidates for CSE. Atomic variables and constants are not candidates. Compound expressions are CSE candidates by default, however when the target is a SAT encoding we exclude all compound expressions that can be encoded as a single SAT literal. This avoids creating a redundant SAT variable that is equal to (or the negation of) another SAT variable, thus improving the encoding. The following expressions are not candidates: $x = c$, $x \neq c$, $x \leq c$, $x < c$, $x \geq c$, $x > c$, $\neg x$ (where x is a decision variable and c is a constant).

The second step of Identical CSE is to iterate through sets of expressions in decreasing size order (line 4). When an expression e is eliminated by CSE, the number of

occurrences of any expressions contained in e is reduced. Therefore eliminating long expressions first may obviate the need to eliminate short expressions. For each set (of size greater than one) of identical expressions a new decision variable *aux* is created, and each of the expressions is replaced with *aux*. One of the expressions e in the set is used to create a new constraint $e = aux$. Crucially the new constraint contains the original object e so it is possible to extract further CSEs from within e.

Prior to running Identical CSE the model is simplified by evaluating all constant expressions and placing it into negation normal form. In addition some type-specific simplifications are performed (eg $x \leftrightarrow \mathrm{true}$ rewrites to x). Commutative expressions (such as sums) are sorted to make some equivalent expressions syntactically identical.

In our previous work we investigated Associative-Commutative CSE (AC-CSE) for constraint solvers [19] and in that context Identical CSE was always enabled. Identical CSE is complementary to AC-CSE.

Active CSE Active CSE extends Identical CSE by allowing non-identical expressions to be extracted using a single auxiliary variable. For example, suppose we have $x = y$ and $x \neq y$ in the model. We can introduce a single Boolean variable a and a new constraint $a \leftrightarrow (x = y)$, then replace $x = y$ with a and $x \neq y$ with $\neg a$. For solvers that support negation (such as SAT solvers) $\neg a$ can be expressed in the solver input language with no further rewriting, so we have avoided encoding both $x = y$ and $x \neq y$.

The Active CSE algorithm implemented in SAVILE ROW is an extension of Algorithm 1. The algorithm works as follows: for each candidate expression e a simple transformation is applied to it (for example producing $\neg e$). The transformed expression is placed into the normal form and commutative subexpressions are sorted. The algorithm then queries *map* to discover expressions matching the transformed expression.

Active CSE as implemented in SAVILE ROW 1.6.3 applies four transformations: Boolean negation, arithmetic negation, multiply by 2, and multiply by -2. Rendl implemented Boolean negation active CSE in her Tailor system, along with active reformulations based upon De Morgan's laws and Horn clauses [20]. In SAVILE ROW, the use of negation normal form obviates the use of the latter two. To our knowledge MiniZinc [26, 13] does not implement Active CSE.

Associative-Commutative CSE (AC-CSE) Nightingale et al [19] (for finite domains) and Araya et al [1] (for numerical CSP) established the use of AC-CSE for constraint models. To our knowledge neither Tailor [10, 20] nor MiniZinc [26, 13] implement AC-CSE. It exploits the properties of associativity and commutativity of binary operators, such as in sum constraints. For SAT encoding, our approach refines the procedure for AC-CSE given in Nightingale et al. In that procedure, implied sum constraints are added, which are deduced from global constraints in the model, such as all-different and global cardinality. These implied sums are used to trigger AC-CSE. Since large sum constraints are cumbersome to encode in SAT, and can therefore degrade performance, we add a test to check whether the implied sums are modified following AC-CSE. If not, they are deemed not to be useful and removed prior to SAT encoding.

Extended resolution [2] is gaining interest and can be viewed as AC-CSE applied directly to the disjunctive clauses of a SAT formula.

Effects of CSE on the output formula We give a short example of a constraint reformulation and its effect on the SAT encoding. Suppose we have two occurrences of

$x \times y$, both are contained in sums, and $x, y \in \{1 \dots 10\}$. Ordinarily we would create a new auxiliary variable ($a_1, a_2 \in \{1 \dots 100\}$) for each occurrence, and add two new constraints: $x \times y = a_1$ and $x \times y = a_2$. Both a_1 and a_2 would be encoded using just under 200 SAT variables and approximately 400 clauses each. Also, both new constraints would be encoded using 100 clauses each. In contrast, Identical CSE would create a single auxiliary variable for both occurrences of $x \times y$, and there would be one new constraint, saving hundreds of SAT variables and clauses. It is difficult to see how SAT pre/inprocessing rules could identify the structure that was exploited by Identical CSE.

3 Experimental Evaluation

Our goal is to investigate whether reformulations performed on a constraint problem instance are beneficial when the problem instance is solved by encoding to SAT and using a state-of-the-art SAT solver. To achieve this we need to ensure that the baseline encoding to SAT is sensible. Therefore we have used standard encodings from the literature such as the order encoding for sums [28] and support encoding [8] for binary constraints. Also we do not attempt to encode all constraints in the language: several constraint types are decomposed before encoding to SAT. Details are given in the SAVILE ROW tutorial 1.6.3 appendix A [18].

In our experiments we compare four configurations of SAVILE ROW: *Basic*, which includes the default options of unifying equal variables, filtering domains and aggregation; *Identical CSE*, which is Basic plus Identical CSE; *Identical & Active CSE*, which is Basic plus the two named CSE algorithms, and *Identical, Active & AC-CSE*, which is Basic plus all three CSE algorithms. Our benchmark set is the set of example problems included with SAVILE ROW 1.6.3 [18]. There are 49 problem classes including common benchmark problems such as EFPA [11] and car sequencing [6] as well as less common problems such as Black Hole solitaire [9]. In total there are 492 problem instances.

Experiments were run with 32 processes in parallel on a machine with two 16-core AMD Opteron 6272 CPUs at 2.1 GHz and 256 GB RAM. We used the SAT solver Lingeling [12] which was winner of the Sequential, Application SAT+UNSAT track of the SAT competition 2014. We downloaded lingeling-ayv-86bf266-140429.zip from http://fmv.jku.at/lingeling/. We used default options for Lingeling so inprocessing was switched on. All times reported include SAVILE ROW time and Lingeling's reported time, and are a median of 10 runs with 10 different random seeds given to Lingeling. A time limit of one hour was applied. We used a clause limit of 100 million, and for instances that exceeded the clause limit we treated them as if they timed out at 3600s (to allow comparison with others). Of 492 instances, 7 reached the clause limit with *Basic* and 6 with the other configurations.

Summary Plots for Full Set of Benchmarks In Figures 1–2 we present a summary view of the performance of our CSE methods over our full set of 49 benchmark problem classes. Figure 1 (upper left) compares the basic encoding with identical CSE. On easier problem instances CSE has a limited effect, but as problem difficulty increases so does the potential of identical CSE to reduce search effort very significantly - in some cases by over 20 times. There are a small number of outliers among the harder instances where identical CSE degrades overall performance. We conjecture that this is due to

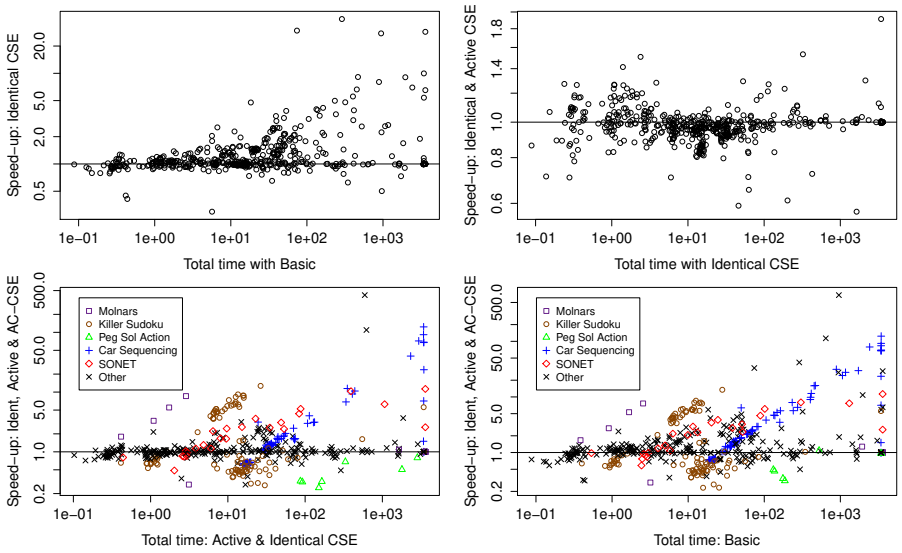

Fig. 1. Identical CSE vs Basic (upper left), Active & Identical CSE vs Identical CSE (upper right), Identical, Active & AC-CSE vs Identical & Active CSE (lower left), same vs Basic (lower right).

the change in problem structure affecting the heuristics of Lingeling. The degradation effect is limited compared with the potential for a large speedup and the number of outliers is small. The geometric mean speed-up is 1.24.

In Figure 1 (upper right) we compare identical CSE alone with identical CSE combined with active CSE. The results show that this additional step is largely neutral or incurs a very small overhead of performing the active CSE checks, but that there are a number of occasions where active CSE significantly enhances identical CSE. Again, there are a small number of outliers where performance is significantly degraded, which we again believe to be due to a bad interaction with the SAT solver search strategy. The geometric mean speed-up is 0.98, indicating a very small average slow-down.

Figure 1 (lower left and right) plots the utility of AC-CSE. In some cases we see a very considerable improvement in performance, however there are also cases where performance is degraded. Five notable problem classes have been separated in the plots. Of these, Killer Sudoku is the most ambiguous, with clusters of instances both above and below the break-even line. For Car Sequencing and SONET, some of the easier instances are below the break-even line, but the more difficult instances exhibit a speed-up. Peg Solitaire Action is degraded on all seven instances. Molnars exhibits a speed up with one exception. Over all instances the geometric mean speed-up is 1.24.

To partly explain these results, we measured the size of the formula produced with and without AC-CSE. Figure 2 has the same y-axis as Figure 1 (lower left) but with a different x-axis: the ratio of the number of variables in the SAT formula. Values of x above 1 indicate that applying AC-CSE has reduced the number of SAT variables. For Killer Sudoku, there is a clear link between the number of SAT variables in the formula

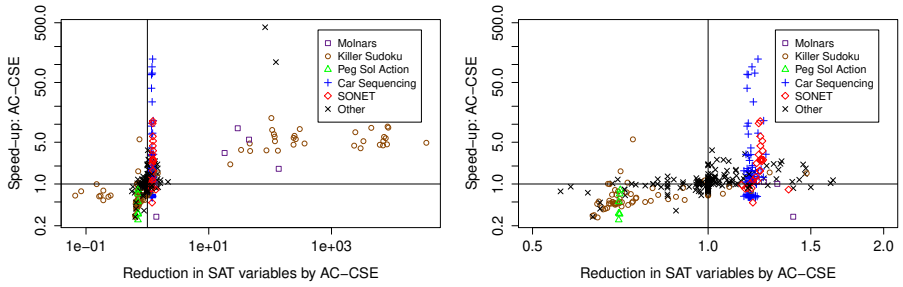

Fig. 2. Identical, Active & AC-CSE vs Identical & Active CSE plotted against reduction in SAT variables. The plot on the right is a subset of the left.

and the speed up quotient. It is also worth noting that for all instances of Peg Solitaire Action applying AC-CSE both increases the number of SAT variables and degrades performance. On the other hand Car Sequencing and SONET show no correlation between speed up quotient and reduction of SAT variables, indicating that the number of SAT variables alone is a very coarse measure of difficulty.

Case study 1: Car Sequencing Our ESSENCE′ model of Car Sequencing [24] uses a sequence of integer variables $x[1 \ldots n]$ to represent the sequence of cars on a production line. For each option (to be fitted at a station on the production line) we have a limit on the proportion of cars: at most p of any q adjacent cars may have the option installed so as not to overload the station. To model this we employ overlapping sums of length q containing $x[i] \in S$, where S is the set of car classes that have the option installed, and i is an index within the subsequence of length q.

The number of each car class to build is enforced with a global cardinality constraint [17] on x. Also, for each option we know how many cars require that option in total (t) thus we add a further *implied* constraint: $\sum_{i=1}^{n}(x[i] \in S) = t$. We experiment with the 80 instances used in Nightingale [17]. Identical and Active CSE are both able to extract the expressions $x[i] \in S$ that appear in several sum constraints, avoiding multiple SAT encodings of the same set-inclusion constraint. Figure 3 (left) plots the time improvement of Active CSE compared with the Basic encoding. The improvement is substantial and increases with the difficulty of the problem instances.

AC-CSE is able to extract common subexpressions among the sum constraints for a given option. The p of q constraints overlap with each other and also with the implied constraint for the option. Figure 1 (lower left) plots the time improvement of adding AC-CSE to identical and active CSE. The additional improvement is substantial, with many instances becoming solvable within one hour and a peak speed up of over 100 times. With the Basic encoding 13 of the 80 instances time out at one hour. In contrast, when combining Identical, Active and AC-CSE we found that only two instances timed out. The other 11 are solved within one hour, most with very substantial speed-ups.

Case study 2: SONET The SONET problem [16, 25] is a network design problem where each node is installed on a set of *rings* (fibre-optic connections). If two nodes are required to be connected, there must exist a ring on which they are both installed. We use

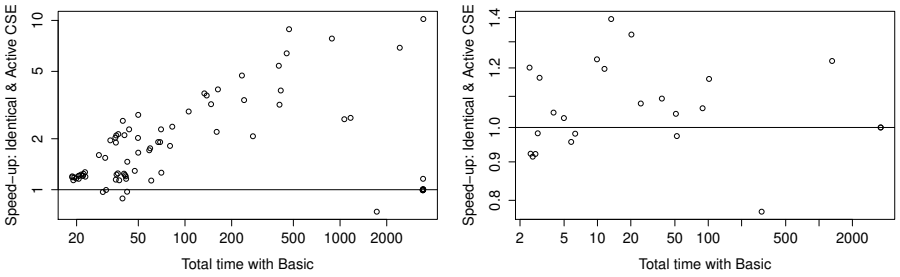

Fig. 3. Identical & Active CSE vs Basic: Car Sequencing (left) and SONET (right).

the simplified SONET problem where each ring has unlimited data capacity (Section 3 of [25]). Rings are indistinguishable so we use lexicographic ordering constraints to order the rings in non-decreasing order. This is an optimisation problem: the number of node-ring connections is minimised. The problem formulation and set of 24 instances are exactly as described in Nightingale et al [19].

Figure 3 (right) compares Identical and Active CSE to the Basic encoding on this problem class. The initial formulation of SONET contains no identical or active common subexpressions, however each decomposition of a lexicographic ordering constraint has identical subexpressions that are exploited by Identical CSE, causing the modest gains seen in the plot. There are four groups of constraints in SONET: the objective function, the constraints ensuring nodes are connected when required, a constraint for each ring limiting the number of nodes, and the symmetry breaking constraints. Apart from symmetry breaking all constraints are sums and all three groups overlap, therefore AC-CSE is successful on this problem as shown in Figure 1 (lower left).

4 Conclusion

Common Subexpression Elimination has proven to be a valuable tool in the armoury of reformulations applied to constraint models, however hitherto there has only been limited success in applying CSE to SAT formulations [29]. We have shown how CSE can be used to improve SAT formulations derived through an automated encoding process from constraint models. Our approach has the advantage that it can identify and exploit structure present in a constraint model that is subsequently obscured by the encoding process, while still taking advantage of powerful SAT solvers. The result is a method that, when applicable, can produce a very significant reduction in search effort.

We have evaluated our approach on a wide range of benchmark problems. On some instances we observed improvements of SAT solver speed of over 50 times. On the car sequencing problem, for example, the peak speed increase is over 100 times. With the basic approach, 13 of 80 car sequencing instances could not be solved in one hour, whereas with the full CSE approach only two instances could not be solved.

Acknowledgements We wish to thank the EPSRC for funding this work through grants EP/H004092/1 and EP/M003728/1, and Christopher Jefferson for helpful discussions.

References

1. Araya, I., Neveu, B., Trombettoni, G.: Exploiting common subexpressions in numerical CSPs. In: Principles and Practice of Constraint Programming (CP 2008). pp. 342–357. Springer (2008)
2. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning sat solvers. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (2010)
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of Computer Aided Verification. pp. 171–177 (2011)
4. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS Press (2009)
5. Cocke, J.: Global common subexpression elimination. ACM Sigplan Notices 5(7), 20–24 (1970)
6. Dincbas, M., Simonis, H., Van Hentenryck, P.: Solving the car-sequencing problem in constraint logic programming. In: Proceedings of the 8th European Conference on Artificial Intelligence (ECAI 1988). pp. 290–295 (1988)
7. Eén, N., Biere, A.: Effective preprocessing in sat through variable and clause elimination. In: Theory and Applications of Satisfiability Testing. pp. 61–75. Springer (2005)
8. Gent, I.P.: Arc consistency in SAT. In: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI 2002). pp. 121–125 (2002)
9. Gent, I.P., Jefferson, C., Kelsey, T., Lynce, I., Miguel, I., Nightingale, P., Smith, B.M., Tarim, S.A.: Search in the patience game 'black hole'. AI Communications 20(3), 211–226 (2007)
10. Gent, I.P., Miguel, I., Rendl, A.: Tailoring solver-independent constraint models: A case study with Essence′ and Minion. In: Proceedings of the Seventh Symposium on Abstraction, Reformulation, and Approximation (SARA 2007). pp. 184–199 (2007)
11. Huczynska, S., McKay, P., Miguel, I., Nightingale, P.: Modelling equidistant frequency permutation arrays: An application of constraints to mathematics. In: Proceedings of Principles and Practice of Constraint Programming (CP 2009). pp. 50–64 (2009)
12. Järvisalo, M., Heule, M.J., Biere, A.: Inprocessing rules. In: Automated Reasoning, 6th International Joint Conference, IJCAR 2012, pp. 355–370. Springer (2012)
13. Leo, K., Tack, G.: Multi-pass high-level presolving. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI) (2015), to appear
14. Marques-Silva, J.: Practical applications of boolean satisfiability. In: 9th International Workshop on Discrete Event Systems (WODES 2008). pp. 74–80 (2008)
15. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th annual Design Automation Conference. pp. 530–535. ACM (2001)
16. Nightingale, P.: CSPLib problem 056: Synchronous optical networking (SONET) problem. `http://www.csplib.org/Problems/prob056`
17. Nightingale, P.: The extended global cardinality constraint: An empirical survey. Artificial Intelligence 175(2), 586–614 (2011)
18. Nightingale, P.: Savile Row, a constraint modelling assistant. `http://savilerow.cs.st-andrews.ac.uk/` (2015)
19. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I.: Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In: 20th International Conference on Principles and Practice of Constraint Programming (CP 2014). pp. 590–605. Springer (2014)
20. Rendl, A.: Effective Compilation of Constraint Models. Ph.D. thesis, University of St Andrews (2010)

21. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier (2006)
22. Shang, Y., Wah, B.W.: A discrete lagrangian-based global-search method for solving satisfiability problems. Journal of global optimization 12(1), 61–99 (1998)
23. Shlyakhter, I., Sridharan, M., Seater, R., Jackson, D.: Exploiting subformula sharing in automatic analysis of quantified formulas. In: Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003) (2003), poster
24. Smith, B.: CSPLib problem 001: Car sequencing. `http://www.csplib.org/Problems/prob001`
25. Smith, B.M.: Symmetry and search in a network design problem. In: 2nd International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2005). pp. 336–350 (2005)
26. Stuckey, P.J., Tack, G.: Minizinc with functions. In: Proceedings of 10th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2013). pp. 268–283 (2013)
27. Subbarayan, S., Pradhan, D.K.: NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In: Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005). pp. 276–291 (2005)
28. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. Constraints 14(2), 254–272 (2009)
29. Yan, Y., Gutierrez, C., Jeriah, J.C., Bao, F.S., Zhang, Y.: Accelerating SAT solving by common subclause elimination. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015). pp. 4224–4225 (2015)